Generic Fault-Tolerance Mechanisms Using the Concept of Logical Execution Time

Christian Buckl, Matthias Regensburger, Alois Knoll, Gerhard Schrott
Department of Informatics
Technische Universität München
Garching b. München, Germany
{buckl,regensbu,knoll,schrott}@in.tum.de

Abstract

Model-based development has become state of the art in software engineering. Unfortunately, the used code generators often focus on the pure application functionality. Features like automatic generation of fault-tolerance mechanisms are not covered. One main reason is the inadequacy of the used models. An adequate model must have amongst others explicit execution semantics and must be suited to support replica determinism and automatic state synchronization. These requirements are fulfilled when using the concept of logical execution time, a time-triggered approach. This approach hides the implementation details like the physical execution from the user, In contrast to other time-triggered paradigms. Within this paper, we present a solution to exploit this concept to realize major fault-tolerance mechanisms in a generic way[1].

1 Introduction

Model-based design has become state of the art in software engineering. Especially the existence of diverse tools for automatic code generation like Matlab/Simulink or SCADE is very attracting. Particularly for the domain of safety-critical applications, where the developer are typically application domain experts with less background in programming fault-tolerant real-time systems [11], the possibility to provide extensive code generation is crucial.

Nevertheless, the code generation abilities of existing tools cover only the functional aspects of the applications like control functions. However, major parts, especially of fault-tolerant embedded systems, are related to *system aspects*. We understand by this term all non-functional aspects related to the distribution of the embedded system and the need for fault-tolerance: process management, scheduling, inter-process communication, communication within the distributed system and the fault-tolerance mechanisms. These aspects are in general not addressed by existing tools and have to be implemented manually.

Due to the non-existence of adequate tools, the development of safety-critical systems is very expensive and many systems are implemented without appropriate fault-tolerance mechanisms. Examples are systems for robot control, off-shore wind mill control or factory automation. Typically, only the most essential safety mechanisms like emergency stop are implemented in such systems. Other mechanisms that could raise the reliability and safety are not implemented due to the high development overhead. The goal of our work is therefore to provide a tool that allows the addition of fault-tolerance mechanisms with minimal overhead. This can be achieved by intensive automatic code generation.

One reason for the non-existence of code generation tools is the absence of adequate models with an explicit semantic. The widely used unified modeling language UML for example, lacks the precision and rigor needed for code generation [7]. Especially, the wish to generate automatically code for fault-tolerance mech-

[1]This work is funded by the German Federal Ministry of Education and Research BMBF under grant 01ISF12A

0-7695-3054-0/07 $25.00 © 2007 IEEE
DOI 10.1109/PRDC.2007.14

anism poses several strict requirements on the application, like replica determinism [13].
One possibility to cope with the need for replica determinism is to use the concept of logical execution times [9], a time-triggered approach. This approach has several advantages: on the one hand, the system behaves deterministic, due to the absence of race conditions. On the other hand, there are previously known points in time, when the fault-tolerance mechanisms like voting can be executed. Within the time-triggered architecture, these characteristics were already exploited to realize fault-tolerance mechanisms at communication level. The time-triggered protocol [16] provides different services like predictable communication with small latency, clock synchronization and a membership service. Errors affecting one electronic control unit (ECU) can be detected automatically, if the errors result in missing or corrupted messages. More detailed error detection must be implemented manually by the developer, e.g. by implementing a distributed voting.
The main contribution of this paper is to point out a way to automatically generate such mechanisms at system level. More explicitly, the developer can specify within our model-based tool the fault-tolerance mechanisms that should be applied. The faults, we intend to cover, comprise amongst others transient or permanent hardware defects, implementation errors, if N-Version programming [1] is used, networking errors and timing violations. In addition to fault-tolerance mechanisms, our code generation also covers other system-level aspects like process management, scheduling, communication and I/O operations.
The remainder of this paper is organized as follows: an overview of our tool and its architecture is given in Sec. 2. Section 3 gives an introduction to the concept of logical execution times and defines our meta-model. Subsequent, the realization of different popular fault-tolerance in the context of time-triggered systems is discussed in Sec.4. Finally, Sec.5 concludes this paper and provides a summary of the planned future work.

2 Code Generation

Before talking about the generic fault-tolerance algorithms, it is necessary to describe the approach for code generation. Our tool covers the generation of all system level aspects like process management, communication and fault-tolerance mechanisms. Thus, it can be used in addition to well-established tools that cover the generation of code at function level.
Due to the vast heterogeneity of platforms[2] used in embedded systems, it is not possible to implement a code generator that covers a priori all possible platforms. Rather, the code generator must be designed in a way that it is easy to extend both regarding the meta-model used for code generation, as well as regarding the code generation ability.
By using a template-based code generator[4], we can achieve this extensibility. The input of the code generator is on the one hand the model and on the other the application dependent code, e.g. the control functions. The code generation is performed using templates. A single template offers an application independent solution for a particular aspect of the system, e.g. scheduling, and is designed for a specific platform. Within the code generation process, the generator core analysis the model, selects appropriate templates and adapts the templates to the application model. A detailed description of this technique can be found in [2]. The result of the code generation process is an optimized run-time system tailored to the application-specific requirements.

3 Meta-Model Using the Time-Triggered Paradigm

In this section, we will explain the core aspects of the meta-model used for the code generation. Because several aspects must be covered by the model, we use a combination of different sub models: a hardware architecture model, a software architecture model, a fault model and a fault tolerance model. Within this paper, we will only describe the main concepts of the software architecture, the fault and fault tolerance model. A complete overview is given in [3].
As execution model, we are using the concept of logical execution time [9], a time-triggered paradigm. In contrast to standard time-triggered approaches, this concept hides the physical execution of tasks. The developer has rather to specify the points in time, when

[2]We understand the combination of programming language, operating system and hardware as platform.

the tasks are logically started by reading the arguments and stopped by publishing the result. A concrete schedule is generated by our tool. The behavior of the system is deterministic since race conditions are excluded by design. Another big advantage is the existence of previously known points in time, when distributed fault-tolerance mechanisms must be executed. Furthermore, the developer of distributed decision protocols benefits from this knowledge: for each expected message, there exists a time frame when this message arrives in an error-free system. Thus, the design of protocols to gain a consistent view of the system state between all error-free units is simplified.

Also, replica determinism can be achieved by using the knowledge about the execution times [13]: at specific points in time a deterministic behavior of the system is guaranteed, while between these points in time, the process execution and scheduling can be carried out in different ways on the individual units. Of course, this is only true, when each individual unit gets exactly the same input and the same algorithms are used. This contradicts the usage of replicated analog sensors and N-Version programming. To bypass this problem, we allow also the usage of interval voting. In this case, the developer has to define the maximal deviation between two correct values.

The core idea within the software model is the concept of **simple tasks** [8]: tasks are periodically called functions that are **independent** from each other. The task functions consist of sequential code without any synchronization points. Blocking communication and synchronization mechanisms are not allowed during task execution.

The communication between tasks is realized in a time-triggered manner using **ports**. A port is a space in memory with a predetermined size and interpretation. Instead of using local ports assigned to each task, like for example in Giotto [6], ports are specified within our model globally. Thus, the values of the ports represent the current state of the system, which is necessary for automatic state synchronization. At the logical start of a task execution, the values of the ports are read as function parameters, at the logical end of a task execution, the results are written to a set of ports. The values of the ports represent the state of the unit, as already mentioned. Therefore, task functions are not allowed to store state information in internal variables.

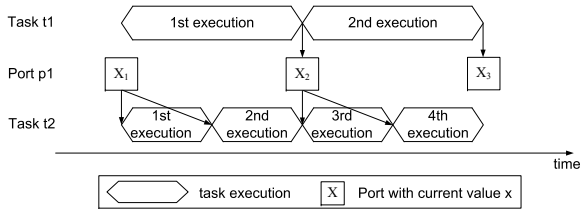

Figure 1. Task and Port Interaction

Rather, the developer has to use additional ports.

Figure 1 shows an example scenario: two tasks t1 and t2 are communicating via a port. In this scenario, the task t2 has twice the frequency of t1. This means that task t2 reads the result of t1 twice, even if task t1 has finished its physical execution on the CPU, before task t2 starts the second execution.

To guarantee the deterministic behavior of the application, the user must declare mechanisms to solve possible conflicts due to simultaneous write accesses on one port. The user can decide to select the median of the values, to calculate the average value or to use local copies of one port. These mechanisms are especially useful, when using redundant components to achieve fault-tolerance.

The interaction with the environment takes place also in a time-triggered manner. All points in time, when **input** and **output** functions are executed, must be specified at compile time. Further concepts (jobs, modes, triggers) allow a more flexible design of the application.

Fault Hypothesis: The selection of the generated fault-tolerance mechanisms is based on the fault model and the selected fault-tolerance mechanisms specified by the developer. One example is the consensus protocol that depends on the reliability of the network.

Within the fault model, the developer can specify the faults that should be tolerated and the fault-containment units, the components that are affected by a specific fault. If possible, the developer can also state the probability of the different faults and the behavior of erroneous units (e.g. fail-silent, fail-hold, etc.).

In general, we do not pose any restrictions on the faults. The system is designed in a way that all kind of faults may occur at task level. Nevertheless, we assume that none of the generated mechanisms can be affected by an erroneous task. This assumption can be legitimated by using an operating system with mem-

ory management unit. The scheduling algorithm can also be designed in a way that erroneous tasks can not affect other tasks on the same unit.
The generated fault-tolerance mechanisms are assumed to be error-free. In case, the ECU is erroneous, we assume the failure mode to be symmetric. This can be legitimated by the fact that only the generated mechanisms can interact directly with the environment. Byzantine failures are excluded by design.

Fault-Tolerance Mechanisms: The fault-tolerance model is used for the specification of fault-tolerance mechanisms that should be used within the system. The basic mechanisms fall into three different categories: error detection, error handling and error recovery. By using these basic mechanisms, typical fault-tolerance patterns like hot-/cold-standby, fault masking, backward recovery and reconfiguration can be implemented. A detailed discussion on the available mechanisms can be found in Section 4.

Example: To illustrate our approach, we want to give a simple example: a PID controller running on a TMR system. The general data flow is depicted in Fig.2. An input function is executed every 1ms and the result is written into the port $Sense$. The task reads the value of this port as well as two other auxiliary ports used for the calculation of the integral part and the derivative part and writes the results to the result port and updates the auxiliary ports. An output function triggers the actuator. To achieve fault-tolerance, the user has to specify three replicas of the input, output and task objects that are each assigned to one control unit. To cope with the simultaneous write access of the redundant items on these ports, the developer can decide to have a local copy of the ports $Sense$ and $Last$ on each control. Thus, he can avoid a communication round before the task execution to obtain a consistent value. The other two ports must be assimilated after each round. Otherwise, small differences within the calibration of the sensor, e.g. an AD converter, can sum up in the port Sum and lead to a disagreement between the different nodes. Within the fault-tolerance model, the developer can then select a voting test to be performed on port $Result$. If a disagreement, e.g. by using interval testing, is detected, the erroneous unit will be excluded from the system. The voting algorithm also decides which unit should perform the output, in case the user has specified that exactly one control unit should perform the output. To allow also an integration of a previously excluded node, the user has to state that the values of the ports Sum and $Last$ have to be copied to the integrating unit.

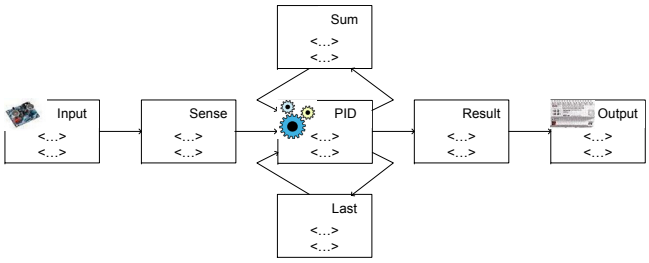

Figure 2. Application Model: a PID Controller

4 Fault-Tolerance Mechanisms

In this section, we will point out possibilities to realize the fault-tolerance techniques within time-triggered systems. Due to space limitations, we can point out only the major mechanisms. All suggestions made are independent from the specific hardware. If some requirements are posed to the hardware architecture, these requirements are stated explicitly. In addition, the realization of the mechanisms depends on the fault hypothesis. The assumptions made are stated wherever necessary.
The first part of this section describes basic requirements and assumptions. Afterwards different tests for error detection and localization are discussed. Mechanisms for error handling/masking and error recovery round up this section.

4.1 Basics

In this subsection, the mechanisms for a basic fault-tolerant distributed system, networking and temporal synchronization, are discussed.

Network: In general, we make no assumptions on the reliability of the used networks. However, since the realization of different mechanisms is dependent on the reliability of the network, the concrete implementation may differ drastically depending on the reliability. Within this paper, we assume two kinds of networks: a reliable network with no message loss, e.g. TTP, but also unreliable protocols like Ethernet.
One assumption that is posed to all used networks is

related to the time-triggered paradigm: we assume a maximal message latency considering a known network load. This can be achieved by using Switched Ethernet or real-time protocols like CAN (controller area network).

Temporal Synchronization: Temporal synchronization is one of the key features required to design a fault-tolerant system. If the communication layer does not provide a service for clock synchronization, an appropriate solution can be realized on top of the network stack exploiting the time-triggered scheme. By means of the expected and the actual arrival time of the individual messages, a logical global clock can be computed [10, 15]. To allow a fault-tolerant temporal synchronization, the majority of nodes must be non-faulty [5].
The algorithm exists of two parts: the arrival time is attached to each incoming message. The unit is allowed to process the message, if the message was received within the expected time frame. Otherwise, a communication error or a temporal synchronization error is assumed. To avoid a corruption of the time stamp, the reception and storage of the network messages must be performed with high priority.
The second part of the algorithm is invoked periodically and checks the deviation of the local to the global clock. For this purpose, all messages within the buffer are analyzed and the difference of the expected reception time to the actual reception time is calculated. If this difference is greater than the tolerated deviation ϵ the message is discarded. Otherwise, the time difference of all regular messages is accumulated and the arithmetic mean is calculated. In addition, the number of valid messages/senders is counted. If the number of senders is lower than the minimal required number of senders, either the local clock or the network is faulty. Another possibility is the presence of more faulty units than tolerated. If enough valid messages are received, the global clock can be adjusted. To avoid permanent adjustments, a threshold ρ is introduced that must be exceeded before the local clock is adjusted.
To achieve an initial temporal synchronization an algorithm similar to the start-up algorithm in TTP/C [16] can be used. It is obvious that the precision of the temporal synchronization is limited by two factors: the maximal network message delay and the precision of the local clock. Within our tests, we achieved maximal synchronization errors below 200 μs using Switched Ethernet and the standard real-time clock in VxWorks.

4.2 Error detection

The first step in fault-tolerant systems is to detect errors. Errors can be detected by using timing tests, hardware tests, test on port values, as well as liveliness tests. We will focus on tests that can be implemented independent from the application or by the use of very little input from the application developer. Within our system, we use the tests to determine the state of the fault containment units. Changes in this state can trigger fault-tolerance actions that are described in the succeeding sections.

Timing Tests: The first obvious tests in time-triggered systems are of course tests related to the timing behavior. In the previous subsection, we already mentioned timing errors related to the clock. Of course, also time violations regarding the task execution can be detected very easily. Within the run-time system, tests are performed to check whether the individual tasks complete their executions on time. In case the WCETs are previously known, scheduling algorithms can be constructed that guarantee time slots for each individual tasks [12]. Thus the effects of timing violations can be restricted to the individual task.

Hardware Tests: Examples for hardware tests can be CPU or memory tests. For frequently used processors, one can include relevant CPU tests as templates. The required test frequency can be determined by analyzing the used safety standard or can be specified by the developer. As memory tests are often not acceptable due to the long execution times of such tests, the user can also specify to store the data redundant within memory.

Tests on Port Values: This class of tests can be divided into relative and absolute tests. Absolute tests can be performed on one unit without knowledge about the states of other redundant units. If the application designer specifies rules about the port values like valid intervals, maximal deviation between two successive port values, these acceptance tests can be performed very easily at runtime.

Algorithm 1 Voting - 1st round

```
1: matrix=initializeMatrix;
2: sendState();
3: while timeout do{
4:     //receive voting messages until timeout
5:     addMessage(buffer,receiveVotingMessages());
6: end while
7: for all pairs of messages i,j in buffer do
8:     result=compareMessages(i,j);
9:     updateMatrix(matrix,result,i,j);
10: end for
```

For the application of relative tests, the states of redundant units must be available. A common problem is the necessity of replica determinism, meaning that all fault-free should behave in the same way. Replica nondeterminism can also be the result of clock synchronization imprecision ε, of N-Version programming or of imprecision of measurement results. Due to these effects, the port values of correct, redundant units are typically situated in a small interval. To overcome this problem, interval decisions can be used for voting. In this case, the application designer has to specify allowed deviations for correct values.
The generation of generic voting algorithms is simplified, as the state of each processing unit is reflected in the port values.
Of course, the realization of voting depends heavily on the assumptions made on the reliability of the network. In this paper, we will assume an unreliable network with at most one communication error (broken link or lost/corrupted message within each communication round). A possible way to achieve a distributed voting decision is depicted in algorithm 1 and 2. The usage of an unreliable network is tolerated by performing the voting in two rounds. If the network is reliable, the second round can be avoided. To avoid unnecessary further network load, voting and normal communication messages are combined. The result of the voting algorithm is stored in an agreement matrix containing the information, whether or not the states of two participating units are consistent with each other. To minimize the network load without endangering the consistency of the decision, within a second round, the results of the first round instead of the complete state are forwarded to the other voting units.

The initial step in the first round is to send the own state to all participating units. Afterwards, the states of the other units are received. This step is limited by time: the maximal time needed for message transmission plus the maximal synchronization error. After the reception, the messages are analyzed: all pairs of states are compared and the result of the comparison is entered in the matrix (typically 0 for disagreement, 1 for agreement). The analysis concludes the first round and each units waits for the begin of the second round. In this round, the agreement matrix is send to the other nodes. Afterwards, the units wait for incoming messages and analyze these messages: all matrices are combined by the OR operation. This approach is motivated by the assumption that message corruption can be detected, e.g. using CRC tests, and leads to a loss of the message. Note that in addition, Byzantine errors were excluded by design. Therefore, a deviation in an entry of the matrix is always caused by the loss of a message. In other words, if one node states the agreement of two units, this statement can be adopted. Note that the result of this voting algorithm is a decision, which result is erroneous. It is not ensured that all units received all redundant results.

Liveliness Tests: The developer has the possibility to state the heartbeat period for a liveliness test. A redundant unit, e.g. in a hot- or cold-standby system, supervises the selected unit and can detect errors. To minimize the overhead, regular actions of the supervised unit that can be detected by the observing unit are used for this test. Only if no such action is performed within the heartbeat period, an additional message is transmitted.

4.3 Error Handling

Changes of the error state of a fault containment unit can trigger diverse mechanisms to mask this error. Errors can be masked without compromising the real-time constraints by using redundant units, e.g. using hot/cold standby or Triple Modular Redundancy (TMR) systems.

Hot and Cold Standby: Hot or cold standby are typical mechanisms to achieve fault-tolerance. Especially in cold standby, one main problem is state synchronization. In time-triggered systems nevertheless, this is a trivial issue similar to check pointing. Since

Algorithm 2 Voting - 2nd round

```
 1: waitForRound2();
 2: sendMatrix(matrix);
 3: while timeout do{
 4:     m=receiveMatrix();//receive results of 1.round
 5:     matrix=or(matrix,m);
 6: end while
 7: for each line i in matrix do
 8:     sum=sumUpLine(i);//check correctness of i
 9:     if sum< ρ then
10:         notifyError(i)
11:         if ( theni==ownId)
12:             ERROR;
13:         end if
14:     end if
15: end for
```

the system state and the functional behavior are separated, the state can be regularly sent to the redundant unit. In case the primary unit fails, the redundant unit can start the execution immediately.
Another problem concerning hot standby is the timely recognition of failures. Since in time-triggered system all behavior is specified in relation to time, an optimal value for the points in time, when to test the active unit, can be determined easily.

TMR Architectures: The basic mechanism to allow the implementation of TMR architectures is voting. In subsection 4.2 the algorithm for voting was presented. The task of failure masking is to get a consistent agreement of the unit that is performing the output. If all correct units are allowed to output, even simultaneously, this is a trivial issue. In this case, all units have to analyze the relevant line in the agreement matrix containing the information about the own agreement state. If enough units, typically the majority, agreed with the own state, the unit can perform the output.
If exact one correct unit has to perform the output, a possibility is that the unit with the lowest ID and with a majority of votes performs the output. Another possibility is that the unit with most votes and the lowest id performs the output.
If there are not enough votes for the own unit, the node will be excluded by the system and should perform repair mechanisms. Since these repair actions are typically application dependent, the possibilities to design generic algorithms are limited. Possibilities are the restart of the system or rebooting the control unit. The final design decision, which actions should be conducted, is made by the application designer.

4.4 Error Recovery

Components within erroneous fault containment units are typically excluded from system operation and perform repair operation. This can be done e.g. by rollback recovery or other typically application dependent mechanisms. After a successful repair, it is often necessary to integrate the unit into the running system.

Rollback Recovery: In case of a transient or intermittent fault, a repeated execution of some function can overcome the error [14]. Instead of executing the same function again, it can be also promising to use some other function or to execute the function on another processing unit. Rollback recovery is a rather simple approach, the only difficulty arise in designing and generating the checkpoints. Since the values of ports represent the state of the unit and ports are updated in a time-triggered manner, it is relatively easy to realize this check pointing. Based on the fault model and the available memory, the set of affected ports, the number of checkpoints and the frequency of the check pointing can be generated. Due to the time-triggered paradigm, problems like domino effects in multiprocessor environments are excluded by design.

Integration of Nodes: A successful state synchronization is the prerequisite of integration. The synchronization is simplified by the concept of global ports. A requirement for an enduring success of the integration is that at transmission time, no relevant tasks are running on the forwarding node. Otherwise, the update of the port values at the end of the task execution could lead to a repeated exclusion of the previously integrated unit. The latter condition can be checked by analyzing the time schedule. A possibility to guarantee the correctness of the received data is to perform voting on the transmitted data, if redundant units are forwarding the current system state to the integration unit. The ports needed for integration can be specified by the application designer within the model.

5 Conclusion

In this paper, we discussed an approach to automatically generate fault-tolerance mechanisms using model-based development. Besides fault-tolerance mechanisms, the presented code generation tool allows also the generation of other system level code, like communication or process management. The tool can be used in addition to existing tools like Matlab/Simulink or SCADE to accelerate the application implementation and to reduce the error rate by the usage of pre-implemented components.

The main contribution of this paper is to point out a way to generate these mechanisms by using the concept of logically execution times. The concept fulfills the posed requirements like replica determinism and provides previously known points in time to perform distributed fault-tolerance mechanisms. By separating the functional behavior, reflected in the concept of tasks, from the state of the system, reflected in the concept of ports, we formed the basis for an automatic realization of voting and integration. Due to the variety of different fault-tolerance mechanisms, we focused on standard mechanisms. However, our approach is also suited for other mechanisms due to the extensibility of our tool.

The approach was tested in several lab applications using hot-redundancy or triple-modular redundancy architectures. The automatic code generation rate reached up to 95% and we achieved sample rates up to 1 kHz [2].

The next step within the project is to apply the tool in an industrial development process to point out the feasibility of the approach. Further research is related to the qualification of the code generator by the German certification authority TÜV and the usage of formal methods to proof the correctness of the generated code, especially the adequacy of selected fault-tolerance mechanisms and architecture concerning the fault-model.

References

[1] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault-tolerance during program execution. In *Proc. COMPSAC'77*, Nov 1977.

[2] C. Buckl, A. Knoll, and G. Schrott. Model-based development of fault-tolerant embedded software. In *Second International Symposium on Leveraging Applications of Fomal Methods, Verification and Validation (IEEE-ISoLA)*, pages 113–120, 2006.

[3] C. Buckl, M. Regensburger, A. Knoll, and G. Schrott. Models for automatic generation of safety-critical real-time systems. In *Second International Conference on Availability, Reliability and Security (ARES 2007)*, pages 580–587. IEEE Computer Society, Apr 2007.

[4] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[5] F. Cristian and C. Fetzer. Probabilistic internal clock synchronization. In *Symposium on Reliable Distributed Systems*, pages 22–31, 1994.

[6] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *First International Workshop on Embedded Software (EMSOFT)*, pages 166 – 184, 2001.

[7] I. Johnson, C. Snook, A. Edmunds, and M. Butler. Rigorous development of reusable, domain-specific components, for complex applications. In *CSDUML'04 - 3rd International Workshop on Critical Systems Development with UML*, 2004.

[8] H. Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Dordrecht, Nederlands, 1997.

[9] H. Kopetz and G. Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112 – 126, Jan. 2003.

[10] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *J. ACM*, 32(1):52–78, 1985.

[11] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1990.

[12] J. W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[13] S. Poledna, A. Burns, A. Wellings, and P. Barrett. Replica determinism and flexible scheduling in hard real-time dependable systems. *IEEE Transactions on Computers*, 49:100–110, Feb. 2000.

[14] D. K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice Hall, 1996.

[15] U. Schmid and K. Schossmaier. Interval-based clock synchronization. *Real-Time Systems*, 12(2):173–228, 1997.

[16] TTTech Computertechnik AG. Time Triggered Protocol TTP/C High-Level Specification Document. 2003.